

Matching Logic

Grigore Rosu

University of Illinois at Urbana-Champaign


Joint work with Andrei Stefanescu and
Chucky Ellison. Started with Wolfram Schulte
at Microsoft Research in 2009

Question

... could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?

Hoare logic

$\{\pi_{\text{pre}}\} \text{code} \{\pi_{\text{post}}\}$



Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L
 - Typically skipped: considered expensive and useless
- Model checkers for L
 - Based on some adhoc encodings of L
- Program verifiers for L
 - Based on some other adhoc encodings of L
- Runtime verifiers for L
 - Based on yet another adhoc encodings of L
- ...

Example of C Program

- What should the following program evaluate to?

```
int main(void) {  
    int x = 0;  
    return (x = 1) + (x = 2);  
}
```

- According to C “standard”, it is **undefined**
- GCC4, MSVC: it returns **4**
GCC3, ICC, Clang: it returns **3**
By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc "prove" it returns **4**

A Formal Semantics Manifesto

- Programming languages must have formal semantics! (period)
- Informal manuals are not sufficient
 - Manuals typically have a formal syntax of the language (in an appendix)
 - Why not a formal semantics as well?

Motivation and Goal

- We are facing a semantic chaos
 - Operational, denotational, axiomatic, etc.
 - Problematic when dealing with large languages
- Why so many semantic styles?
 - Since none of them is ideal, they have limitations
- We want a powerful, unified foundation for programming language semantics and verification
 - One semantics to serve all the purposes!

Minimal Requirements for an Ideal Language Semantic Framework

- Should be **expressive**
 - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be **executable**
 - So we can test it and use it in tools (symb. exec.)
- Should be **modular** (and thus scale)
 - So each feature is defined once and for all
- Should serve as a **program logic**
 - So we can also prove programs correct with it

Current Semantic Approaches

- Structural Operational Semantics
 - Executable; not very modular; not appropriate for verification; only interleaving semantics
- Denotational Semantics
 - Reasonable trade-offs; not very executable (Norish's C semantics is not executable, and factorial(5) crushes Papaspyrou's C semantics); not very good for verification; bad for concurrency; expert knowledge
- Axiomatic Semantics (Floyd/Hoare logic)
 - Good for verification (sort of); not executable; not very expressive (extensions typically needed)
- Etc.

Towards a Better Semantic Approach

Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

- **Expressive**
 - Any logic can be represented in RL (it is reflective)
- **Executable**
 - Quite efficiently; Maude often outperforms SML
- **Modular**
 - Allows rules to only “match” what they need
- Can potentially serve as a **program logic**
 - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs

Rewriting Logic Semantics Project

- Project started jointly with Meseguer in 2003-4
- Idea: Define the semantics of a programming language as a rewrite theory (set of rules)
- Showed that most executable semantics approaches can be framed as rewrite logic semantics (Modular/SmallStep/BigStep SOS, evaluation contexts, continuation-based, etc.)
 - But they still had their inherent limitations
- Appropriate techniques/methodologies needed

The K Framework

- A tool-supported rewrite-based framework for defining programming language semantics
- Inspired from rewriting logic
- Used regularly in teaching undergraduate courses
- Ideas:
 - Represent program configurations as a nested structure of cells (like in the CHAM)
 - Flatten syntax into special computational structures (like in refocusing for evaluation contexts)
 - Define the semantics of each language construct by rules (a small number, typically 1 or 2)

Complete K Definition of KernelC

MODULE KERNELC-SYNTAX

```

IMPORTS PL-ID+PL-INT
Exp ::= DeclId
      | Id
      | Inu
      | Exp + Exp [strict]
      | Exp - Exp [strict]
      | Exp ++
      | Exp == Exp [strict]
      | Exp != Exp [strict]
      | Exp <= Exp [strict]
      | Exp < Exp [strict]
      | Exp < Exp [strict]
      | Exp % Exp [strict]
      | ! Exp
      | Exp && Exp
      | Exp || Exp
      | Exp ? Exp : Exp
      | printf ("%d", Exp) [strict]
      | scanf ("%d", Exp) [strict]
      | scanf ("%d", & Exp)
      | NULL
      | PointerId
      | (int*)malloc(Exp * sizeof(int)) [strict]
      | free(Exp) [strict]
      | * Exp [strict]
      | Exp [ Exp ]
      | Exp - Exp [strict(2)]
      | Id ( Lis[Exp] ) [strict(2)]
      | Id ( )
      | random( Exp ) [strict]
      | random ( )

Sexp ::= { }
      | Exp ; [strict]
      | { SexpLis }
      | if ( Exp ) Sexp
      | if ( Exp ) Sexp else Sexp [strict(1)]
      | while ( Exp ) Sexp
      | return Exp ; [strict]
      | DeclId Lis[DeclId] { SexpLis }
      | #include < SexpLis >

SexpLis ::= SexpLis SexpLis
SexpLis ::= SexpLis
Pgm ::= SexpLis
Id ::= main
PointerId ::= Id
DeclId ::= int Exp
          | void PointerId
SexpLis ::= stdio.h
          | stdlib.h

Lis[Bouom] ::= Lis[Bouom] , Lis[Bouom] [assoc hybrid id ( ) strict]
          | Bouom
Lis[PointerId] ::= Lis[PointerId] , Lis[PointerId] [assoc ditto id ( ) ]
          | Lis[Bouom]
          | PointerId
Lis[DeclId] ::= Lis[DeclId] , Lis[DeclId] [assoc ditto id ( ) ]
          | DeclId
          | Lis[Bouom]
Lis[Exp] ::= Lis[Exp] , Lis[Exp] [assoc ditto id ( ) ]
          | Exp
          | Lis[DeclId]
          | Lis[PointerId]

```

END MODULE

MODULE KERNELC-DESUGARED-SYNTAX

```

IMPORTS KERNELC-SYNTAX
MACRO: E = E ? 0 : 1
MACRO: E1 && E2 = E1 & E2 : 0
MACRO: E1 || E2 = E1 | E2
MACRO: if ( E ) S = if ( E ) S else { }
MACRO: NULL = 0
MACRO: f ( ) = f ( )
MACRO: int * PointerId = int PointerId
MACRO: #include < Sexp > = Sexp
MACRO: E1 [ E2 ] = * E1 + E2
MACRO: scanf ("%d", & E) = scanf ("%d", E)
MACRO: int * PointerId = E = int PointerId = E
MACRO: int X = E = int X ; X = E ;
MACRO: stdio.h = { }
MACRO: stdlib.h = { }

```

END MODULE

MODULE KERNELC-SEMANTICS

IMPORTS K-SHARED
IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:



RULE: $\frac{K}{V} \quad \frac{env}{X \mapsto V}$

RULE: $\frac{K}{X++} \quad \frac{env}{X \mapsto I +_{int} 1}$

RULE: $\frac{K}{X = V} \quad \frac{env}{X \mapsto V}$

RULE: $I_1 + I_2 \rightarrow I_1 +_{int} I_2$
 RULE: $I_1 - I_2 \rightarrow I_1 -_{int} I_2$
 RULE: $I_1 \& I_2 \rightarrow I_1 \&_{int} I_2$ when $I_2 \neq_{int} 0$
 RULE: $I_1 < I_2 \rightarrow Bool2int (I_1 <_{int} I_2)$
 RULE: $I_1 < I_2 \rightarrow Bool2int (I_1 <_{int} I_2)$
 RULE: $I_1 = I_2 \rightarrow Bool2int (I_1 =_{int} I_2)$
 RULE: $I_1 != I_2 \rightarrow Bool2int (I_1 !=_{int} I_2)$
 RULE: $! ? \rightarrow if (_) _ else _$
 RULE: $if (I) _ _ else S \rightarrow S$ when $I =_{int} 0$
 RULE: $if (I) S _ else _ \rightarrow S$ when $!_{bool} I =_{int} 0$

RULE: $\frac{K}{while (E) S} \quad \frac{env}{if (E) \{ S \} while (E) S _ else \{ \}}$

PRINT RULE: $\frac{K}{printf ("%d", I)} \quad \frac{env}{S +_{string} int2string (I) +_{string} " "}$

READ-GLOBAL RULE: $\frac{K}{scanf ("%d", N)} \quad \frac{env}{N \mapsto \frac{I}{I +_{int} 1}}$

READ-LOCAL RULE: $\frac{K}{scanf ("%d", X)} \quad \frac{env}{X \mapsto \frac{I}{I +_{int} 1}}$

RULE: $V \rightarrow _$
 RULE: $\{ S \} \rightarrow S$
 RULE: $\{ \} \rightarrow _$
 RULE: $S \ S \rightarrow S _ S$

RULE: $\frac{K}{int X XI (S)} \quad \frac{env}{X \mapsto int X XI (S)}$

RULE: $\frac{K}{void X XI (S)} \quad \frac{env}{S \mapsto return void ;}$

Letsem ::= Id * Map * K
 RULE: $\frac{K}{X (VI) \wedge K} \quad \frac{env}{eraseKLabel (int _ , XI) \mapsto VI}$
 RULE: $\frac{K}{X \mapsto int X XI (S)} \quad \frac{env}{X \mapsto Env * K}$

CONTEXT: int $\rightarrow \square$
 RULE: $\frac{K}{int X} \quad \frac{env}{X \mapsto undo F}$

RULE: $\frac{K}{return V ; \wedge _} \quad \frac{env}{V}$

RULE: $\frac{K}{V \wedge _ \wedge K} \quad \frac{env}{Env} \quad \frac{stack}{_ \mapsto Env * K}$

RULE: $\frac{K}{(int*)malloc (N * sizeof(int))} \quad \frac{env}{N' \mapsto N}$
 RULE: $\frac{K}{N' \dots N +_{nat} N' \mapsto undo F} \quad \frac{env}{N' \mapsto N +_{nat} N'}$

RULE: $\frac{K}{free (N)} \quad \frac{env}{N \mapsto N'}$
 RULE: $\frac{K}{random ()} \quad \frac{env}{randomRandom (N')}$

RULE: $\frac{K}{random (I)} \quad \frac{env}{N' \mapsto N +_{nat} 1}$

RULE: $\frac{K}{srandom (I)} \quad \frac{env}{void}$

CONTEXT: $_ \rightarrow \square$
 CONTEXT: $_ \mapsto _ ++$
 Val ::= Inu
 Exp ::= Val
 K ::= Lis[DeclId]
 Lis[Exp] ::= Lis[Exp]
 Lis[PointerId] ::= Lis[PointerId]
 Pgm ::= SexpLis
 String
 restore (Map)
 undo F
 KRule ::= Lis[Val]
 Lis[K] ::= Nat .. Nat
 Rule: $N_1 \dots N_k \mapsto _ +_{nat} (_)$
 Rule: $N_1 \dots N_k \mapsto N +_{nat} N_1 \dots N_k$
 Lis[Val] ::= Lis[Val] , Lis[Val] [assoc ditto id ()]
 Lis[Exp] ::= Lis[Val]

END MODULE

```

MODULE KERNELC-SYNTAX
IMPORTS PL-ID+PL-INT
Exp ::= Declfd
| Id
| Int
| Exp + Exp [strict]
| Exp - Exp [strict]
| Exp ++
| Exp == Exp [strict]
| Exp != Exp [strict]
| Exp <= Exp [strict]
| Exp < Exp [strict]
| Exp % Exp [strict]
| ! Exp
| Exp && Exp
| Exp || Exp
| Exp ? Exp : Exp
| printf (" %d", Exp ) [strict]
| scanf ("%d", Exp ) [strict]
| scanf ("%d", &Exp )
| NULL
| Pointerfd
| ( int * ) malloc ( Exp * sizeof ( int ) ) [strict]
| free ( Exp ) [strict]
| * Exp [strict]
| Exp - Exp [strict(2)]
| Id + Lis ( Exp ) + [strict(2)]
| Id ( )
| random ( Exp ) [strict]
| random ( )

Sum ::= ( )
| Exp ; [strict]
| { Sum Lis }
| if ( Exp ) Sum
| if ( Exp ) Sum else Sum [strict(1)]
| while ( Exp ) Sum
| return Exp ; [strict]
| Declfd Lis [Declfd] { Sum Lis }
| #include < Sum Lis >

Sum.fis ::= Sum.fis Sum.fis
Sum ::= Sum
Pgm ::= Sum Lis
Id ::= main
Pointerfd ::= Id
| * Pointerfd [ditto]
Declfd ::= int Exp
| void Pointerfd
Sum.fis ::= stdio.h
| stdlib.h

Lis ( Bouom ) ::= Lis ( Bouom ) , Lis ( Bouom ) [assoc hybrid id ( ) strict]
| ( )
| Lis ( Pointerfd ) ::= Lis ( Pointerfd ) , Lis ( Pointerfd ) [assoc ditto id ( )]
| Lis ( Bouom )
| Pointerfd
Lis ( Declfd ) ::= Lis ( Declfd ) , Lis ( Declfd ) [assoc ditto id ( )]
| Declfd
| Lis ( Bouom )
Lis ( Exp ) ::= Lis ( Exp ) , Lis ( Exp ) [assoc ditto id ( )]
| Exp
| Lis ( Declfd )
| Lis ( Pointerfd )

```

```

MODULE KERNEL.C-DESUGARING-SYNTAX
IMPORTS KERNEL.C-SYNTAX

MACRO:
   $E = E ? 0 : 1$ 
MACRO:
   $E_1 \&\& E_2 = 1 ? E_2 : 0$ 
MACRO:
   $E_1 \vee E_2 = 1 ? E_1 : E_2$ 
MACRO:
  if (E) S; else if (E2) S2 else {}
MACRO:
  NULL = 0
MACRO:
  I () = I ()
MACRO:
  int * Pointerid = int Pointerid
MACRO:
  #include <Sumus> = Sumus
MACRO:
  E1, E2 = E1 + E2
MACRO:
  scanf ("%s%d", &E) = scanf ("%s%d", E)
MACRO:
  int * Pointerid = E = int Pointerid = E
MACRO:
  int X = E; = int X; X = E;
MACRO:
  stdio.h = {}
MACRO:
  stdlib.h = {}

END MODULE

```

RULE:

$$\begin{matrix} \text{X} \\ \text{V} \end{matrix} \begin{matrix} \text{k} \\ \text{mny} \end{matrix} \rightarrow \text{V}$$

RULE:

$$\begin{matrix} \text{X} \\ \text{X} \end{matrix} \begin{matrix} \text{k} \\ \text{mny} \end{matrix} \rightarrow \text{I}$$
[illegible]
$$Exp ::=$$

$$| Exp = Exp \text{ [strict(2)]}$$

PRINT RULE:

```

    (k)
    printf ("%d", I )
    void
  
```

$S \leftarrow \text{string} \text{ int2string } (I) \leftarrow \text{string} \rightarrow T$

READ-GLOBAL RULE:

```

    (k)
    scanf ("%d", N )
    void
  
```

$N \leftarrow \frac{\text{mem}}{T}$

$\frac{\text{in}}{T}$

READ-LOCAL RULE:

```

    (k)
    scanf ("%d", X )
    void
  
```

$X \leftarrow \frac{\text{s/m}}{T}$

$\frac{\text{in}}{T}$

RULE: $V \rightarrow \bullet_{\text{in}}$

RULE: $\{ Str \} \rightarrow Str$

RULE: $\{ \} \rightarrow \bullet_{\text{in}}$

RULE: $Se \ Se \rightarrow Se \sim Se$

```

RULE:  $\frac{\text{void}}{\text{void}}$ 

CONTEXT: + □ = −
CONTEXT: + □ ++

Val ::= Int
      | void
Exp ::= Val
      | K ::= Lst[Declid]
      | Lst[Exp]
      | Lst[Polworld]
      | Pgm
      | SomeLst
      | String
      | restore ( Map )
      | undefn r
KRetain ::= Lst[Lst]
Lst[K] ::= Nat .. Nat
RULE:  $N_1 \dots N_1 \rightarrow_{\text{static}} [K]$ 

RULE:  $N_1 \dots n_{N_1} N \rightarrow N \ N_1 \dots N$ 

Lst[Exp] ::= Lst[Val] , Lst[Val] [assoc ditto id ( )]
      | Val
Lst[Val] ::= Lst[Val]

```

Complete K Definition of KernelC

MODULE KERNELC-SYNTAX

IMPORTS PL-ID+PL-INT

```
Exp ::= DeclId
| Id
| Inu
| Exp + Exp [strict]
| Exp - Exp [strict]
| Exp ++
| Exp == Exp [strict]
| Exp != Exp [strict]
| Exp <= Exp [strict]
| Exp < Exp [strict]
| Exp % Exp [strict]
| ! Exp
| Exp && Exp
| Exp || Exp
| Exp ? Exp : Exp
| printf (" %d", Exp ) [strict]
| scanf (" %d", Exp ) [strict]
| scanf (" %d", & Exp )
| NULL
| PointerId
  ( (int*)malloc ( Exp * sizeof(int) ) ) [strict]
  free ( Exp ) [strict]
  * Exp [strict]
  Exp { Exp }
  Exp = Exp [strict(2)]
  Id { Lis { Exp } } [strict(2)]
  Id { }
  random ( Exp ) [strict]
  random ()
```

```
Some ::= { }
| Exp ; [strict]
| { SomeLis }
  if ( Exp ) Some
  while ( Exp ) Some
  return Exp ; [strict]
  DeclId Lis { DeclId } { SomeLis }
  #include < SomeLis >
```

```
SomeLis ::= SomeLis SomeLis
| Some
Pgm ::= SomeLis
Id ::= main
PointerId ::= Id
DeclId ::= int Exp
| void PointerId
SomeLis ::= stdio.h
| stdlib.h
Lis { Bouom } ::= Lis { Bouom } , Lis
| Bouom
| Lis { PointerId } ,
  Lis { Bouom }
  PointerId
Lis { DeclId } ::= Lis { DeclId } , Lis
  DeclId
  Lis { Bouom }
Lis { Exp } ::= Lis { Exp } , Lis { Exp }
| Exp
  Lis { DeclId }
  Lis { PointerId }
```

END MODULE

MODULE KERNELC-DESUGARED

IMPORTS KERNELC-SYNTAX

```
MACRO: 1 E = E ? 0 : 1
MACRO: E1 && E2 = E1 ? E2 : 0
MACRO: E1 || E2 = E1 ? 1 : E2
MACRO: if ( E ) S1 = if ( E ) S1
MACRO: NULL = 0
MACRO: f ( ) = f ( )
MACRO: int * PointerId = int PointerId
  #include < SomeLis > = SomeLis
MACRO: E1 { E2 } = * E1 + E2
MACRO: scanf (" %d", & E ) = scanf
  int * PointerId = E = int PointerId
  int X = E ; int X ; X = E ;
MACRO: stdio.h = { }
MACRO: stdlib.h = { }
```

END MODULE

MODULE KERNELC-SEMANTICS

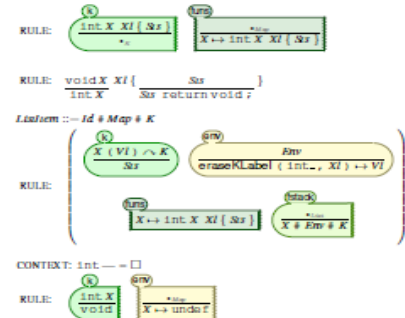
IMPORTS K-SHARED

IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM

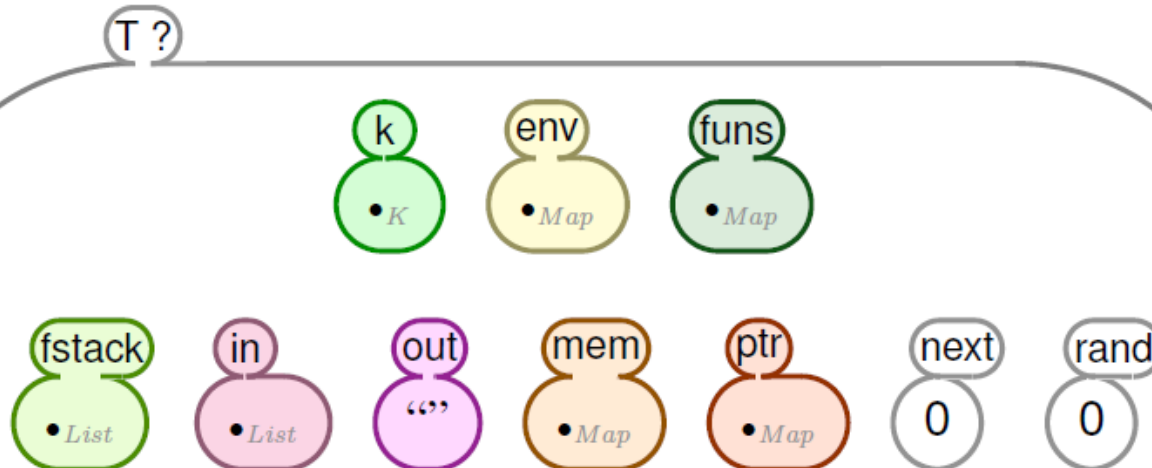
CONFIGURATION:



RULE:



Configuration given as a nested cell structure.
Leaves can be sets, multisets, lists, maps, or syntax



RULE: Sx Sx → Sx ∼ Sx

Lis { Exp } ::= Lis { Val }

END MODULE

Complete K Definition of KernelC

```

MODULE KERNELC-SYNTAX
IMPORTS PL-ID+PL-INT
Exp ::= DeclId
| Id
| Inu
| Exp + Exp [strict]
| Exp - Exp [strict]
| Exp ++
| Exp == Exp [strict]
| Exp != Exp [strict]
| Exp <= Exp [strict]
| Exp < Exp [strict]
| Exp % Exp [strict]
| ! Exp
| Exp && Exp
| Exp || Exp
| Exp ? Exp : Exp
| printf ("%d", Exp) [strict]
| scanf ("%d", Exp) [strict]
| scanf ("%d", & Exp)
| NULL
| PointerId
| (int*)malloc( Exp * sizeof(int)) [strict]
| free( Exp ) [strict]
| * Exp [strict]
| Exp [ Exp ]
| Exp - Exp [strict(2)]
| Id ( Lis{Exp} ) [strict(2)]
| Id ()
| random( Exp ) [strict]
| random ()

Sexp ::= {}
| Exp ; [strict]
| { Sexp Lis }
| if ( Exp ) Sexp else Sexp [strict(1)]
| while ( Exp ) Sexp
| return Exp ; [strict]
| DeclId Lis{DeclId} { Sexp Lis }
| #include< Sexp Lis >

Sexp Lis ::= Sexp Lis Sexp Lis
Pgm ::= Sexp Lis
Id ::= main
PointerId ::= Id
DeclId ::= int Exp
| void PointerId
Sexp Lis ::= stdio.h
| stdlib.h
Lis{Boxom} ::= Lis{Boxom} , Lis{Boxom} [assoc hybrid id () strict]
| ()
Lis{PointerId} ::= Lis{PointerId} , Lis{PointerId} [assoc ditto id () ]
| Lis{Boxom}
| PointerId
Lis{DeclId} ::= Lis{DeclId} , Lis{DeclId} [assoc ditto id () ]
| DeclId
| Lis{Boxom}
Lis{Exp} ::= Lis{Exp} , Lis{Exp} [assoc ditto id () ]
| Exp
| Lis{DeclId}
| Lis{PointerId}
END MODULE

```

```

MODULE KERNELC-DESUGARED-SYNTAX
IMPORTS KERNELC-SYNTAX
MACRO: 1 E = E ? 0 : 1
MACRO: E1 && E2 = E1 ? E2 : 0
MACRO: E1 || E2 = E1 ? 1 : E2
MACRO: if ( E ) S = if ( E ) S else {}
MACRO: NULL = 0
MACRO: f () = f ()
MACRO: int * PointerId = int PointerId
| #include< Sexp > = Sexp
MACRO: E1 [ E2 ] = * E1 + E2
MACRO: scanf ("%d", & E) = scanf ("%d", E)
MACRO: int * PointerId = E = int PointerId = E
MACRO: int X = E ; = int X ; X = E ;
MACRO: stdio.h = {}
MACRO: stdlib.h = {}
END MODULE

```

MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:



RULE: $\frac{K}{X \mapsto V}$ $\frac{env}{X \mapsto V}$

RULE: $\frac{K}{X++} \frac{env}{X \mapsto I}$ $\frac{env}{I \mapsto I + \text{int } 1}$

RULE: $\frac{K}{X = V} \frac{env}{X \mapsto V}$

RULE: $I_1 + I_2 \rightarrow I_1 + I_2$

RULE: $I_1 - I_2 \rightarrow I_1 - I_2$

RULE: $I_1 \& I_2 \rightarrow I_1 \& I_2$ when $I_2 \neq 0$

RULE: $I_1 < I_2 \rightarrow \text{Bool2int}(I_1 < I_2)$

RULE: $I_1 < I_2 \rightarrow \text{Bool2int}(I_1 < I_2)$

RULE: $I_1 \neq I_2 \rightarrow \text{Bool2int}(I_1 \neq I_2)$

RULE: $I_1 \neq I_2 \rightarrow \text{Bool2int}(I_1 \neq I_2)$

RULE: $! I \rightarrow \text{if } (I) \text{ else } _$

RULE: $\text{if } (I) \text{ else } S \rightarrow S$

RULE: $\text{if } (I) \text{ S else } _ \rightarrow S$

RULE: $\frac{K}{\text{while } (E) \&}$ $\frac{env}{\text{if } (E) \{ S \text{ while } (E) \& S \}}$

PRINT RULE: $\frac{K}{\text{printf } ("%d", I)}$ $\frac{env}{\text{void}}$

READ-GLOBAL RULE: $\frac{K}{\text{scanf } ("%d", \text{void})}$

READ-LOCAL RULE: $\frac{K}{\text{scanf } ("%d", \& \text{void})}$

RULE: $V \mapsto _$

RULE: $\{ S \} \rightarrow S$

RULE: $\{ \} \rightarrow _$

RULE: $S \& S \rightarrow S \& S$

RULE: $\frac{K}{\text{int } X \text{ XI } [S]}$ $\frac{env}{X \mapsto \text{int } X \text{ XI } [S]}$

RULE: $\frac{\text{void } X \text{ XI } [S]}{\text{int } X}$ $\frac{env}{S \text{ return void ;}}$

Letsem ::= Id * Map * K

RULE: $\frac{K}{X (VI) \wedge K}$ $\frac{env}{\text{eraseKLabel } (\text{int } _, \text{XI}) \mapsto VI}$

RULE: $\frac{env}{X \mapsto \text{int } X \text{ XI } [S]}$ $\frac{stack}{X \& Env \& K}$

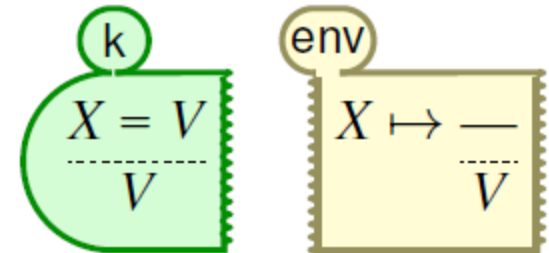
CONTEXT: int ::= □

RULE: $\frac{K}{\text{int } X}$ $\frac{env}{X \mapsto \text{undo } F}$

RULE: $\frac{K}{\text{return } V ; \wedge _}$ $\frac{env}{V}$

RULE: $\frac{K}{V \wedge _ \wedge K}$ $\frac{env}{Env}$ $\frac{stack}{\& Env \& K}$

Semantic rules given contextually



$\langle k \rangle X = V \Rightarrow V \langle _ / k \rangle$

$\langle env _ \rangle X \mapsto _ \Rightarrow (_ \Rightarrow V) \langle _ / env \rangle$

Lis{Exp} ::= Lis{Val}
END MODULE

K Scales

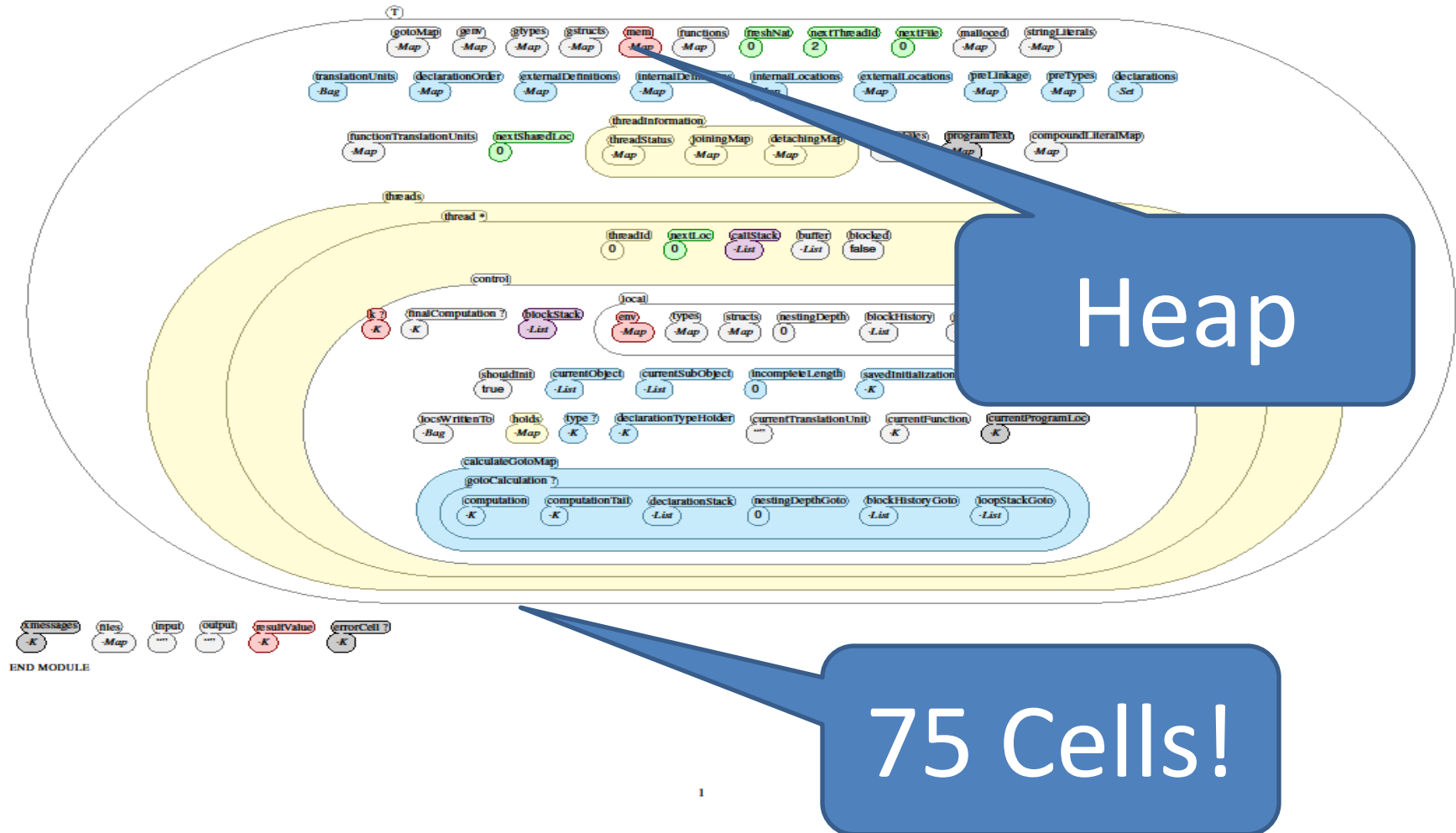
Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.

The K Configuration of C

MODULE C-CONFIGURATION
IMPORTS C-SYNTAX
IMPORTS COMMON-C-CONFIGURATION



Statistics for the C definition

- Syntactic constructs: 173
- Total number of rules: 812
- Total number of lines: 4688
- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test – passed 96% so far)
- The most complete formal C semantics
- Took more than 1 year to define ...
 - Wouldn't it be uneconomical to redefine it in each tool?

Executable Semantics are Useful

- Compiler certification (Leroy's talk)
- Help language designers
- K semantics are currently compiled into
 - Maude, for execution, debugging, model checking
 - Latex, for human inspection and understanding
 - Soon to OCAML, for fast execution
- Can we use K semantics for program verification?
 - E.g., we want to use the C semantics unchanged for program verification; defining an alternative (axiomatic) semantics is very inconvenient and error prone!

Matching Logic = K + FOL

- A logic for reasoning about configurations
- Formulae
 - FOL over configurations, called **patterns**
 - Configurations are allowed to contain variables
- Models
 - Ground configurations
- Satisfaction
 - **Matching** for configurations, plus FOL for the rest

Examples of Patterns I

- x is bound to 5 and is the only variable in environment

$$\langle x \mapsto 5 \rangle_{\text{env}}$$

- x is bound to 5 in the current environment

$$\exists \rho \left(\langle x \mapsto 5, \rho \rangle_{\text{env}} \right)$$

- x is bound to a non-negative value

$$\exists a \exists \rho \left(\langle x \mapsto a, \rho \rangle_{\text{env}} \wedge a \geq 0 \right)$$

- x and y hold equal positive values

$$\exists a \exists \rho \left(\langle x \mapsto a, y \mapsto a, \rho \rangle_{\text{env}} \wedge a > 0 \right)$$

Examples of Patterns II

- x and y are aliased

$$\exists a \exists \rho \exists u \exists \sigma (\langle x \mapsto a, y \mapsto a, \rho \rangle_{\text{env}} \langle a \mapsto u, \sigma \rangle_{\text{heap}})$$

- x and y not aliased, and x points to larger value

$$\exists a \exists b \exists \rho \exists u \exists v \exists \sigma (\langle x \mapsto a, y \mapsto b, \rho \rangle_{\text{env}} \langle a \mapsto u, b \mapsto v, \sigma \rangle_{\text{heap}} \wedge u > v)$$

- x points to a list containing sequence A

$$\exists a \exists \rho \exists \sigma (\langle x \mapsto a, \rho \rangle_{\text{env}} \langle \text{list}(a, A), \sigma \rangle_{\text{heap}})$$

Examples of Patterns III

- x points to sequence A , and the reversed sequence A has been output

$$\exists a \exists \rho \exists \sigma \exists \omega \left(\langle x \mapsto a, \rho \rangle_{\text{env}} \langle \text{list}(a, A), \sigma \rangle_{\text{heap}} \langle \omega, \text{rev}(A) \rangle_{\text{out}} \right)$$

- **untrusted()** can only be from **trusted()**

$$\exists s_1 \exists s_2 \left(\langle \text{untrusted}() \rangle_k \langle s_1, \text{trusted}(), s_2 \rangle_{\text{fstack}} \right)$$

- Read/Write datarace (simplified)

$$\exists X \exists a \left(\langle X \cdots \rangle_k \langle X = a \cdots \rangle_k \right)$$

Matching Logic vs. Separation Logic

- Matching logic achieves separation naturally, through matching at the structural (term) level, not through special logical connectives (*)
- Matching logic realizes separation at all levels of the configuration, not only in the heap; recall that the heap was only 1 out of the 75 cells in C's def.
- Matching logic stays within FOL, while separation logic extends FOL
 - Thus, we can use the existing SMT solvers, etc.

Matching Logic as a Program Logic

- Hoare style - **not recommended**

$$\{\pi_{\text{pre}}\} \text{ code } \{\pi_{\text{post}}\}$$

– One has to redefine the PL semantics – **impractical**

- Rewriting (or K) style – **recommended**

$$\textit{left}[\text{code}] \rightarrow \textit{right}$$

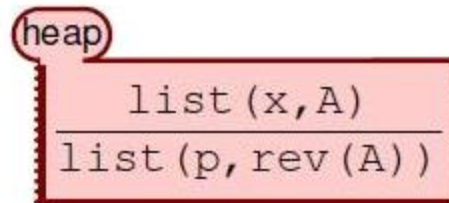
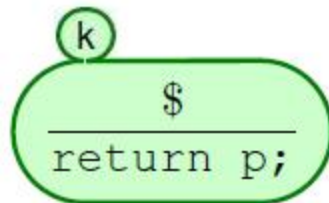
– One can reuse existing K semantics – **very good**

Example – Reversing a list

```

struct listNode* reverse(struct listNode *x)
{
    struct listNode *p;
    struct listNode *y;
    p = 0 ;
    while(x) {
        y = x->next;
        x->next = p;
        p = x;
        x = y;
    }
    return p;
}
    
```

- What is the K semantics of the reverse function?
- Let \$ be its body



$\langle k \rangle \ \$ \Rightarrow \text{return } p; \ \langle /k \rangle$

$\langle \text{heap_} \rangle \ \text{list}(x, A) \Rightarrow \text{list}(p, \text{rev}(A)) \ \langle _ / \text{heap} \rangle$

Partial Correctness

- We have two rewrite relations on configurations
 - given by the language K semantics; **safe**
 - given by specifications; **unsafe**, has to be proved
- Idea (simplified for deterministic languages):
 - Pick **left** → **right**. Show that always **left** → (**→** ∪ **→**)* **right** modulo matching logic reasoning (between rewrite steps)
- Theorem (soundness):
 - If **left** → **right** and “**config** matches **left**” such that **config** has a normal form for →, then “**nf(config)** matches **right**”

MatchC Tool DEMO

(through slides)

Semantic Execution

John Regehr and his team included the K semantics of C as part of his CSMITH tool chain, to make sure that the generated C programs are defined

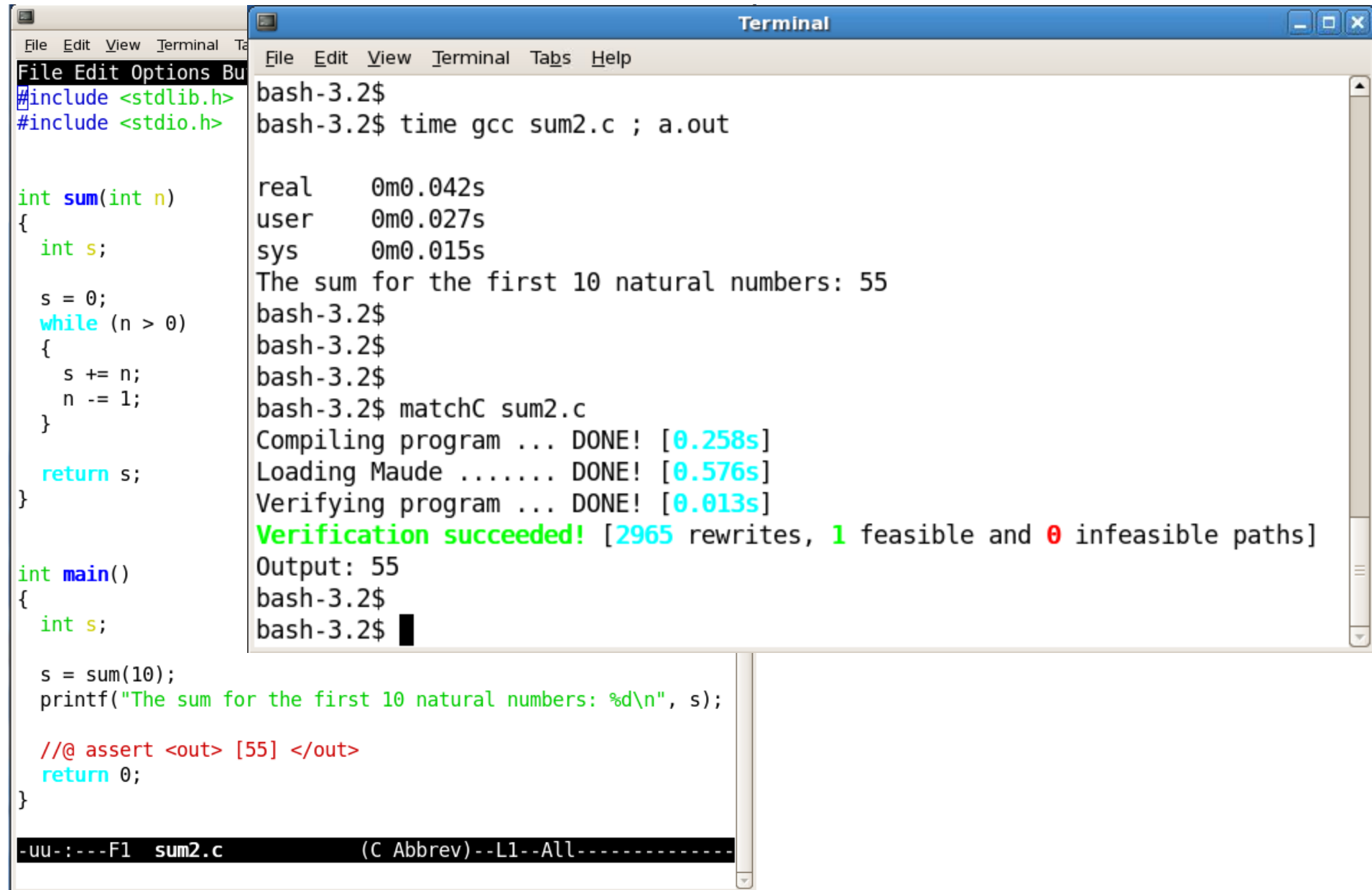
```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help
#include <stdlib.h>
#include <stdio.h>

struct listNode {
    int val;
    struct listNode *next;
};

int main()
{
    struct listNode *x;
    x = (struct listNode *) malloc(sizeof(struct listNode));
    printf("%d\n", x->val);
    return 0;
}

bash-3.2$ time ./undefined.c
real    0m0.052s
user    0m0.022s
sys     0m0.018s
0
bash-3.2$
bash-3.2$
bash-3.2$ matchC undefined.c
Compiling program ... DONE! [0.207s]
Loading Maude ... DONE! [0.576s]
Verifying program ... DONE! [0.003s]
Verification failed! [174 rewrites, 0 feasible and 0 infeasible paths]
Output:
Generating error .... DONE! [0.155s]
Check undefined.kml for the complete output.
bash-3.2$
```

Assertion Checking



The image shows a code editor window on the left and a terminal window on the right. The code editor contains a C program named `sum2.c` that calculates the sum of the first 10 natural numbers. The terminal window shows the execution of the program, the compilation of the Maude model checker, and the successful verification of the program's assertion.

```
File Edit View Terminal Tabs Help
File Edit Options Bu
#include <stdlib.h>
#include <stdio.h>

int sum(int n)
{
    int s;

    s = 0;
    while (n > 0)
    {
        s += n;
        n -= 1;
    }

    return s;
}

int main()
{
    int s;

    s = sum(10);
    printf("The sum for the first 10 natural numbers: %d\n", s);

    //@ assert <out> [55] </out>
    return 0;
}

-uu-:---F1 sum2.c (C Abbrev)--L1--All-----
```

```
Terminal
File Edit View Terminal Tabs Help

bash-3.2$
bash-3.2$ time gcc sum2.c ; a.out

real    0m0.042s
user    0m0.027s
sys      0m0.015s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum2.c
Compiling program ... DONE! [0.258s]
Loading Maude ..... DONE! [0.576s]
Verifying program ... DONE! [0.013s]
Verification succeeded! [2965 rewrites, 1 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```

Full Verification

```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help
#include <stdlib.h>
#include <stdio.h>

int sum(int n)
//@ rule <k> $ => return (n * (n + 1)) / 2; </k> if n >= 0
{
    int s;

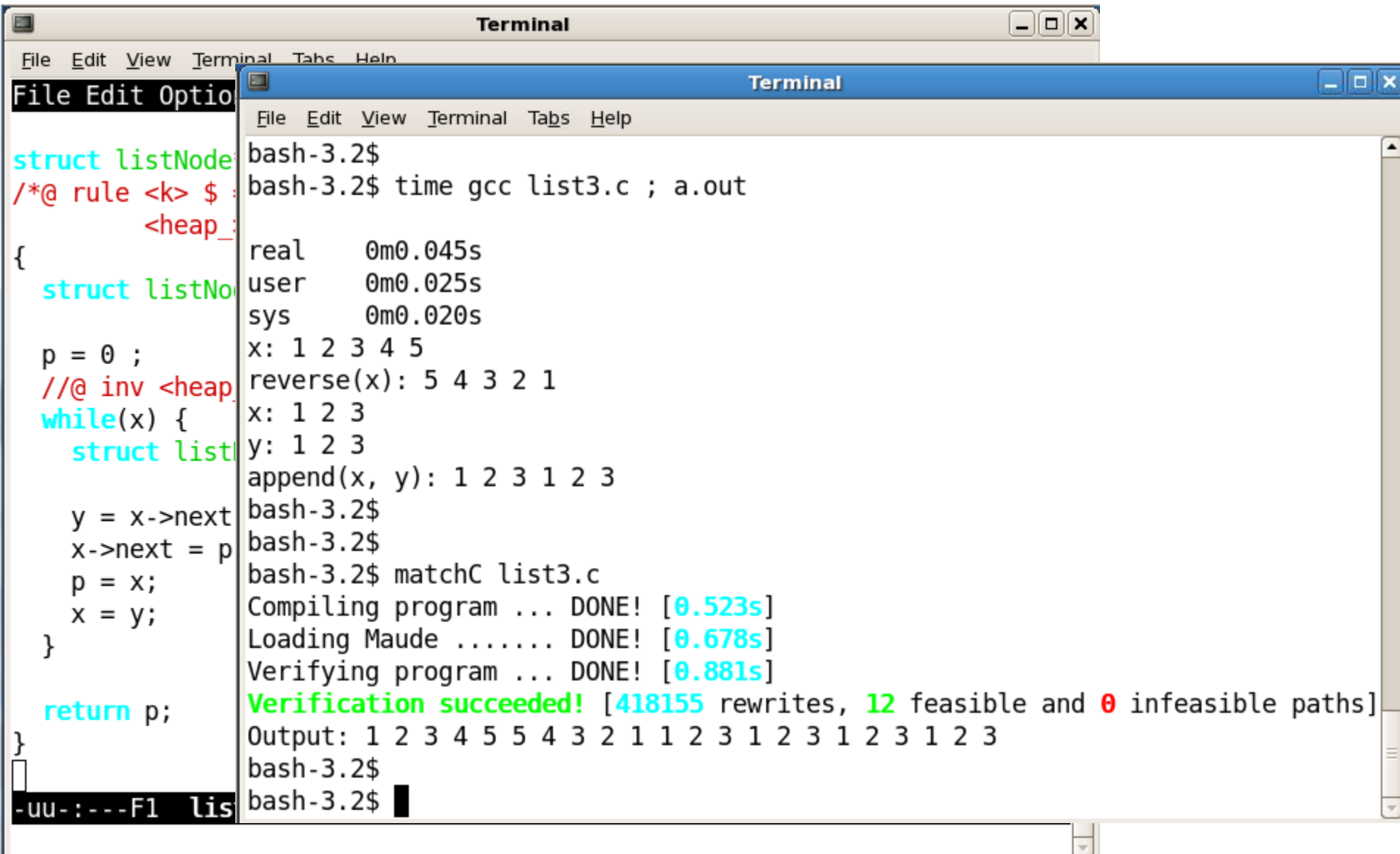
    s = 0;
    //@ inv s = ((old(r
    while (n > 0)
    {
        s += n;
        n -= 1;
    }

    return s;
}
```

```
Terminal
File Edit View Terminal Tabs Help
bash-3.2$
bash-3.2$ time gcc sum3.c ; a.out

real    0m0.042s
user    0m0.023s
sys     0m0.018s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum3.c
Compiling program ... DONE! [0.260s]
Loading Maude ..... DONE! [0.584s]
Verifying program ... DONE! [0.046s]
Verification succeeded! [33083 rewrites, 3 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```


List Examples – Borrowed from SL tools



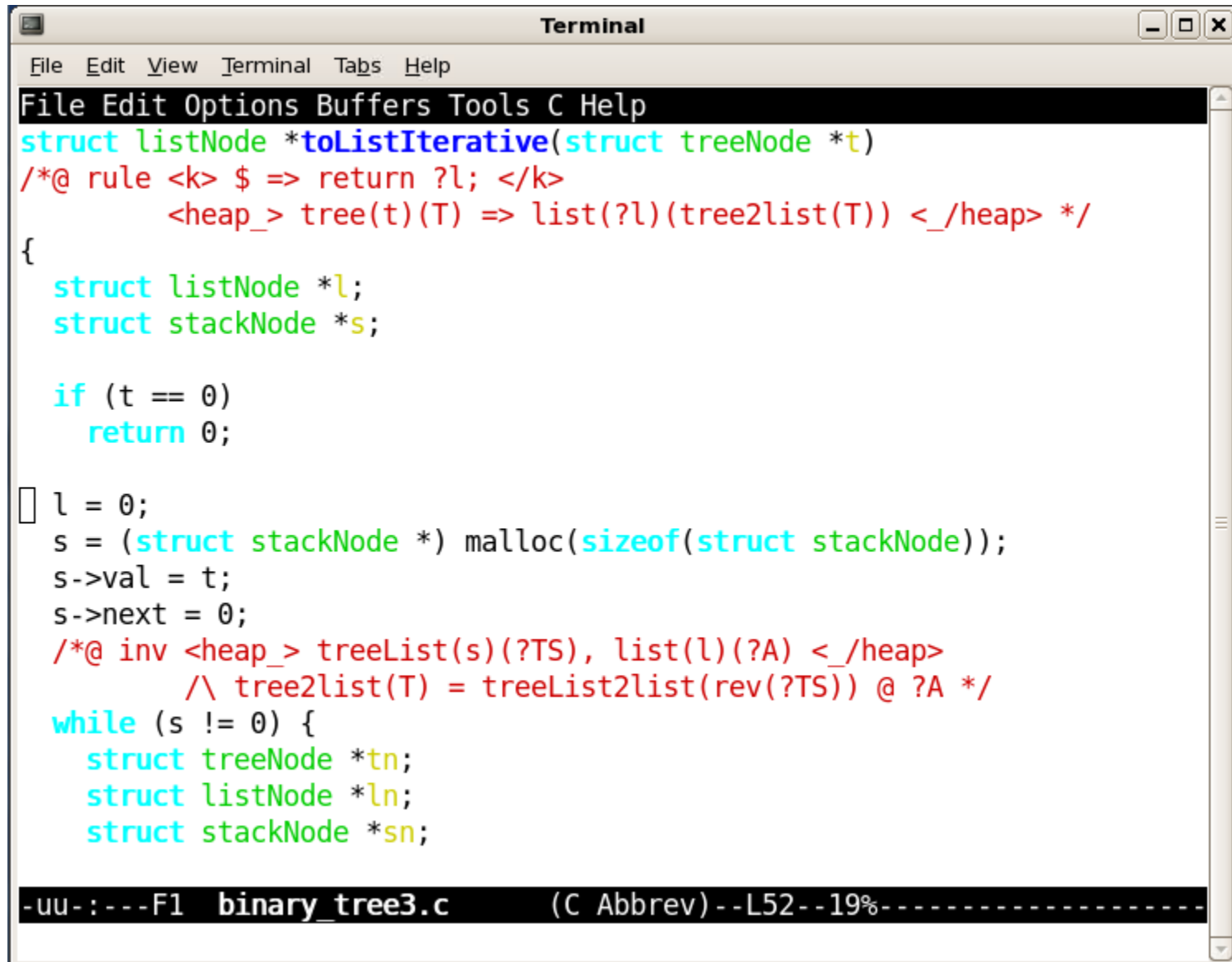
The image shows a terminal window with a blue title bar labeled "Terminal". The window contains the output of a Maude program execution. The program defines a list structure and performs a reversal operation. The output shows the execution time for compiling, loading, and verifying the program, followed by the final list state.

```
bash-3.2$  
bash-3.2$ time gcc list3.c ; a.out  
real    0m0.045s  
user    0m0.025s  
sys     0m0.020s  
x: 1 2 3 4 5  
reverse(x): 5 4 3 2 1  
x: 1 2 3  
y: 1 2 3  
append(x, y): 1 2 3 1 2 3  
bash-3.2$  
bash-3.2$  
bash-3.2$ matchC list3.c  
Compiling program ... DONE! [0.523s]  
Loading Maude ..... DONE! [0.678s]  
Verifying program ... DONE! [0.881s]  
Verification succeeded! [418155 rewrites, 12 feasible and 0 infeasible paths]  
Output: 1 2 3 4 5 5 4 3 2 1 1 2 3 1 2 3 1 2 3 1 2 3  
bash-3.2$  
bash-3.2$
```

On the left side of the terminal window, there is a vertical strip showing the source code of the Maude program. The code defines a list structure and performs a reversal operation. The code is as follows:

```
struct listNode  
/*@ rule <k> $  
  <heap_  
{  
  struct listNo  
  p = 0 ;  
  //@ inv <heap_  
  while(x) {  
    struct list  
    y = x->next  
    x->next = p  
    p = x;  
    x = y;  
  }  
  return p;  
}
```

Beyond Separation Logic Tools



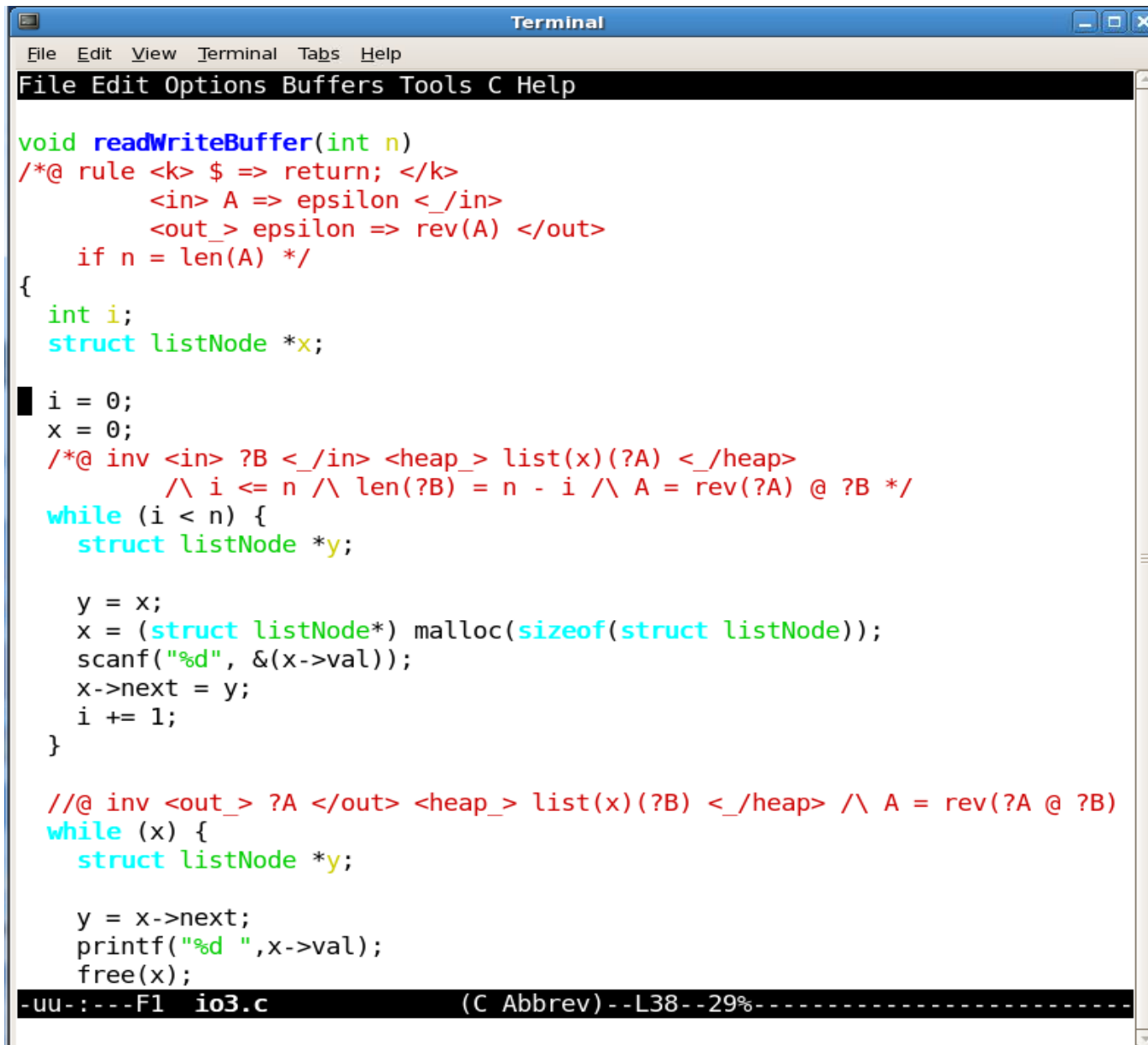
```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help
struct listNode *toListIterative(struct treeNode *t)
/*@ rule <k> $ => return ?l; </k>
    <heap_> tree(t)(T) => list(?l)(tree2list(T)) <_/heap> */
{
    struct listNode *l;
    struct stackNode *s;

    if (t == 0)
        return 0;

    l = 0;
    s = (struct stackNode *) malloc(sizeof(struct stackNode));
    s->val = t;
    s->next = 0;
    /*@ inv <heap_> treeList(s)(?TS), list(l)(?A) <_/heap>
        /\ tree2list(T) = treeList2list(rev(?TS)) @ ?A */
    while (s != 0) {
        struct treeNode *tn;
        struct listNode *ln;
        struct stackNode *sn;
```

-uu-:---F1 binary_tree3.c (C Abbrev)--L52--19%-----

Beyond Separation Logic – I/O



```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help

void readWriteBuffer(int n)
/*@ rule <k> $ => return; </k>
    <in> A => epsilon <_/in>
    <out_> epsilon => rev(A) </out>
    if n = len(A) */
{
    int i;
    struct listNode *x;

    i = 0;
    x = 0;
    /*@ inv <in> ?B <_/in> <heap_> list(x)(?A) <_/heap>
        /\ i <= n /\ len(?B) = n - i /\ A = rev(?A) @ ?B */
    while (i < n) {
        struct listNode *y;

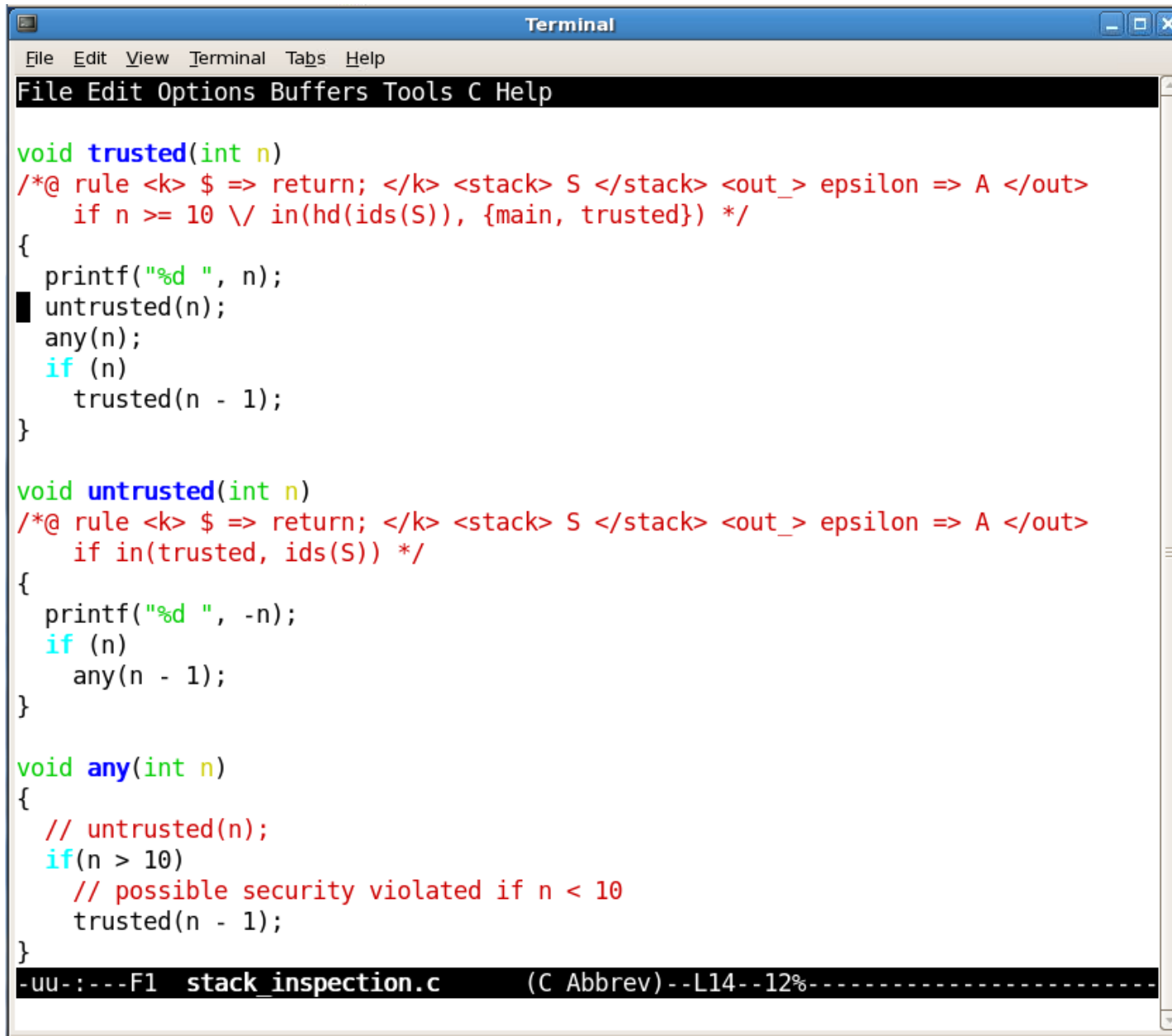
        y = x;
        x = (struct listNode*) malloc(sizeof(struct listNode));
        scanf("%d", &(x->val));
        x->next = y;
        i += 1;
    }

    /*@ inv <out_> ?A </out> <heap_> list(x)(?B) <_/heap> /\ A = rev(?A @ ?B)
    while (x) {
        struct listNode *y;

        y = x->next;
        printf("%d ", x->val);
        free(x);
    }
}
```

-uu-:---F1 io3.c (C Abbrev) --L38--29%-----

Beyond Separation Logic – Stack Inspection

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Tabs, Help) and a toolbar. The main area displays C code for a program called "stack_inspection.c". The code defines three functions: "trusted", "untrusted", and "any". The "trusted" function calls "untrusted" and "any". The "untrusted" function calls "any". The "any" function calls "trusted" if "n" is greater than 10. The code is color-coded: keywords in blue, identifiers in green, and comments in red. The terminal window has a status bar at the bottom showing "-uu:---F1 stack_inspection.c (C Abbrev) --L14--12%-----".

```
void trusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
   if n >= 10 \/\ in(hd(ids(S)), {main, trusted}) */
{
    printf("%d ", n);
    untrusted(n);
    any(n);
    if (n)
        trusted(n - 1);
}

void untrusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
   if in(trusted, ids(S)) */
{
    printf("%d ", -n);
    if (n)
        any(n - 1);
}

void any(int n)
{
    // untrusted(n);
    if(n > 10)
        // possible security violated if n < 10
        trusted(n - 1);
}
```

-uu:---F1 stack_inspection.c (C Abbrev) --L14--12%-----

Conclusions

- Formal semantics is useful and practical!
- One can use an executable semantics of a language *as is* also for program verification
 - As opposed to redefining it as a Hoare logic
- Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools